

# Simplified and Verified: A Second Look at a Proof-Producing Union-Find Algorithm

Lukas Stevens<sup>1,2[0000–0003–0222–6858]</sup> and  
Rebecca Ghidini<sup>1,3,4,5[0009–0009–8117–2659]</sup>

<sup>1</sup> Technical University of Munich, Department of Computer Science,  
Boltzmannstr. 3, Garching, Germany

<sup>2</sup> [lukas.stevens@in.tum.de](mailto:lukas.stevens@in.tum.de) (✉)

<sup>3</sup> École normale supérieure, Département d'informatique de l'ENS,  
CNRS, PSL University, 75005 Paris, France

<sup>4</sup> Inria

<sup>5</sup> [rebecca.ghidini@info.ens.psl.eu](mailto:rebecca.ghidini@info.ens.psl.eu)

**Abstract** Using Isabelle/HOL, we verify a union-find data structure with an explain operation due to Nieuwenhuis and Oliveras. We devise a simpler, more naive version of the explain operation whose soundness and completeness is easy to verify. Then, we prove the original formulation of the explain operation to be equal to our version. Finally, we refine this data structure to Imperative HOL, enabling us to export efficient imperative code. The formalisation provides a stepping stone towards the verification of proof-producing congruence closure algorithms which are a core ingredient of Satisfiability Modulo Theories (SMT) solvers.

**Keywords:** Equivalence closure · Interactive theorem proving · Satisfiability modulo theories · Proof-producing decision procedure

## 1 Introduction

The Union-Find (UF) data structure maintains the equivalence closure of a relation, which is given as a sequence of pairs or, in terms of the UF data structure, UNION operations. It is fundamental to efficiently implement well-known graph algorithms such as Kruskal's [14] minimum spanning-tree algorithm. There, it tracks which vertices belong to the same connected component and are, in that sense, equivalent. Beyond graph algorithms, its applicability extends to the domain of theorem proving as it routinely forms the basis of congruence closure algorithms, which are widely used by Satisfiability Modulo Theories (SMT) solvers. To increase their trustworthiness, current SMT solvers such as cvc5 [3], E [26], Vampire [13], veriT [4], and Z3 [18] can output detailed proofs when they determine an input formula to be unsatisfiable. To produce these proofs, it is crucial to have congruence closure algorithms that efficiently explain why

they consider two terms to be equal. The first such algorithm was presented by Nieuwenhuis and Oliveras [20, 21], who extended the UF data structure with an EXPLAIN operation to obtain a Union-Find-Explain (UFE) data structure. Verifying this data structure in Isabelle/HOL is the focus of our paper.

Why, then, should we verify a data structure that already produces proofs? Our answer is three-fold. (1) While the data structure’s proofs guarantee soundness, we also prove its completeness. (2) Executing code exported from Isabelle depends on a substantial trusted computing base (TCB), including the target language’s compiler and runtime. (3) Whereas SMT solvers typically operate with a large TCB, adding to the TCB of interactive theorem provers is usually discouraged. Proof-producing algorithms — such as those introduced in earlier work by the first author [27] — enable integration without increasing the TCB.

### 1.1 Contributions

We present, to our knowledge, the first formalisation of the UFE data structure as introduced by Nieuwenhuis and Oliveras. In their work, they present two variants of this data structure. Here, we only formalise the first variant, leaving the other for future work. We devise a simpler, more naive version of the EXPLAIN operation, for which soundness and completeness is easier to prove. Then, we prove the original version of the EXPLAIN operation to be extensionally equal to the simple one. Based on an existing formalisation of the UF data structure by Lammich [15], we develop a more abstract formalisation of the data structure, hiding implementation details. Finally, we refine the UFE data structure to Imperative HOL [5] using Lammich’s [15] separation logic framework, enabling generation of efficient imperative code.

The formalisation is available online.<sup>6</sup> Since everything is verified, we omit proofs and focus on outlining the structure of the formalisation.

### 1.2 Related Work

Efficient implementations of the UF data structure have been known for a long time. In particular, Galler and Fisher [9] represent the data structure as a forest of rooted trees and propose the union-by-size heuristic, which gives  $\mathcal{O}(\log n)$  running time for UNION and FIND for a data structure over  $n$  elements. Another heuristic, called path compression, was presented by Aho et al. [1]. Analysing the complexity of the data structure when combining both heuristics turned out to be challenging, but Tarjan [28] and Tarjan and van Leeuwen [29] eventually proved an amortised running time of  $\mathcal{O}(n + m \alpha(m + n, n))$  for a sequence of at most  $n - 1$  UNION and  $m$  FIND operations where  $\alpha$  is the inverse Ackermann function. This means that any one operation runs in almost constant time, amortised.

While the paper on the UFE data structure [20] is widely cited, there is limited follow-up literature on this data structure. It does, however, form the

---

<sup>6</sup> <https://doi.org/10.5281/zenodo.15557955>

basis of proof-producing congruence closure algorithms, which are crucial in the field of SMT solving. There, they remain an active area of research; for example, when we are interested in efficiently finding small proofs [8].

The literature of verified algorithms and data structures is vast so we refer to a survey [22] for an overview. Focusing on the UF data structure, there is a formalisation in Coq [7], where the amortised complexity is analysed by Charugueraud and Pottier [6] in a separation logic with time credits. Similarly, in Isabelle, there is a formalisation of the data structure [15] that was later extended with a complexity analysis by Haslbeck and Lammich [11]. More recently, there is formalisation by Guttman [10] taking a relation-algebraic view.

### 1.3 Notation

Isabelle/HOL [23] conforms to everyday mathematical notation for the most part. We establish notation and in particular some essential data types together with their primitive operations that are specific to Isabelle/HOL.

We write  $t::'a$  to specify that the term  $t$  has the type ' $a$ ' and ' $a \Rightarrow b$ ' for the space of total functions from type ' $a$ ' to type ' $b$ '.

Sets with elements of type ' $a$ ' have the type ' $a$  set'. The cardinality of a set  $A$  is denoted by  $|A|$ .

We use ' $a$  list' to describe the type of lists, which are constructed using the empty list [] or the infix cons constructor (#), and are appended with the infix operator (@). The length of list  $xs$  is denote by  $|xs|$ . The function set converts a list into a set. We write  $xs ! i$  to access the  $i$ -th element of the list  $xs$ .

To represent partial values of type ' $a$ ', we use the type ' $a$  option' with the constructors `None` and `Some`. We also define an order on this type by letting `None` be smaller than `Some` and lifting the order on the underlying type ' $a$ ', i.e. we have that  $(\text{Some } x \leq \text{Some } y) = (x \leq y)$ .

Relations are denoted with the type synonym ' $a$  rel', which expands to  $('a \times 'a) set$ . For a relation  $r$ , `Field r` are those elements that occur as a component of a pair  $p \in r$ . Furthermore, we use  $r^{-1}$  to denote the inverse and  $r^*$  to denote the reflexive transitive closure of  $r$ .

Throughout our formalisation we employ *locales* [2], which are named contexts of types, constants and assumptions about them.

## 2 Basic Union-Find

### 2.1 Background

Given a set of  $n$  elements  $A = \{a_1, \dots, a_n\}$ , the UF data structure keeps track of a partition of  $A$  into disjoint sets  $A_1, \dots, A_k$ , i.e.  $A = A_1 \uplus \dots \uplus A_k$ . Equivalently, one can view the partition as a partial equivalence relation with the equivalence classes  $A_1, \dots, A_k$ . The equivalence relation is partial because  $A::'a$  set might only be a subset of the type ' $a$ '. We initialise the data structure by partitioning

$A$  into singleton sets of elements, so we have that  $A = \{a_1\} \uplus \dots \uplus \{a_n\}$ . Those sets are merged by subsequent UNION operations where UNION  $a_i$   $a_j$  merges the set containing  $a_i$  with the one that contains  $a_j$ . Each set in the partition contains one particular element that serves as its representative. We will denote the representative of an element  $a$  in the UF data structure  $uf$  as `rep-of`  $uf a$ . Accordingly, two elements have the same representative exactly when they belong to the same set in the partition. For any element  $a_i$ , the FIND operation returns its representative `rep-of`  $uf a_i$ .

The data structure can be implemented as a forest of rooted trees where each tree encodes an equivalence class. The edges of a tree in the forest are directed towards the root, which is the representative of the corresponding equivalence class. To preserve this invariant, we initialise the forest with  $n$  vertices but without any edges and, for every UNION of  $a_i$  and  $a_j$ , we add a directed edge from `rep-of`  $uf a_i$  to `rep-of`  $uf a_j$  to the forest.

We encode such a forest as a list  $l$  of length  $n$ , where at each index  $i$  of  $l$ , we save the index of the parent of the element  $a_i$ , denoted by  $l ! i$ . If  $a_i$  is a root, then the list stores  $i$  itself at index  $i$ , i.e.  $l ! i = i$ .

## 2.2 In Isabelle/HOL

The UF algorithm was formalised in Isabelle/HOL by Lammich [15]. The code can be found in an entry [16] of the Archive of Formal Proofs (AFP).<sup>7</sup> Lammich defines a function `rep-of`, which, as described above, follows the parent pointers until we arrive at the root, where the parent pointer is self-referential.

```
rep-of :: nat list ⇒ nat ⇒ nat
rep-of l i = let pi = l ! i in if pi = i then i else rep-of l pi
```

Looking closely at this definition, we see that this function is only well-defined for some inputs  $l$  and  $a$ : for every element  $a < |l|$ , its parent must be in the list, i.e. we must have  $l ! a < |l|$ , and the parent pointers must be cycle-free in order for the function to terminate. Functions in Isabelle/HOL must be total, so Isabelle introduces a constant `rep-of-dom` ::  $\text{nat list} \times \text{nat} \Rightarrow \text{bool}$  that characterises the inputs for which `rep-of` terminates. Then, it adds `rep-of-dom` ( $l$ ,  $a$ ) as a premise to the defining equation of `rep-of`. The intuition above is cast into a predicate `ufa-invar` that defines such well-formed lists  $l$ .

```
ufa-invar :: nat list ⇒ bool
ufa-invar l = ∀ i < |l|. rep-of-dom (l, i) ∧ l ! i < |l|
```

Building on the formalisation, we define the abstract data type (ADT) `ufa` as the set of all  $l::\text{nat list}$  that satisfy `ufa-invar`  $l$ .

```
typedef ufa = {l | ufa-invar l}.
```

---

<sup>7</sup> The code is in the theory file `Examples/Union_Find.thy`.

This introduces a new type without any predefined operations. To equip it with functionality, we lift the operations on the underlying list due to Lammich [15] to the ADT using Isabelle's lifting infrastructure [12], yielding (1)  $\text{ufa-}\alpha :: \text{ufa} \Rightarrow (\text{nat} \times \text{nat}) \text{ set}$ , (2)  $\text{ufa-rep-of} :: \text{ufa} \Rightarrow \text{nat} \Rightarrow \text{nat}$ , (3)  $\text{ufa-init} :: \text{nat} \Rightarrow \text{ufa}$ , and (4)  $\text{ufa-union} :: \text{ufa} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ufa}$ . Their meaning is the following:

- (1)  $\text{ufa-}\alpha \text{ uf}$  is the partial equivalence relation represented by  $\text{uf}$ ,
- (2)  $\text{ufa-rep-of } \text{uf } x$  is the representative of the equivalence class containing  $x$ ,
- (3)  $\text{ufa-init } n$  initialises the data structure with  $n$  elements with each element being its own representative, and
- (4)  $\text{ufa-union } \text{uf } x \text{ } y$  returns a UF data structure where the equivalence classes of  $x$  and  $y$  are merged. This is implemented by updating the underlying list at index  $\text{rep-of } l \text{ } x$  to  $\text{rep-of } l \text{ } y$ .

Formally, the above operations fulfil the properties stated below:

- $\text{ufa-rep-of } \text{uf } x = \text{ufa-rep-of } \text{uf } y \longleftrightarrow (x, y) \in \text{ufa-}\alpha \text{ uf}$  if  $\{x, y\} \subseteq \text{Field } (\text{ufa-}\alpha \text{ uf})$ ,
- $\text{ufa-}\alpha \text{ } (\text{ufa-init } n) = \{(x, x) \mid x < n\}$ , and
- $\text{ufa-}\alpha \text{ } (\text{ufa-union } \text{uf } x \text{ } y) = \text{per-union } (\text{ufa-}\alpha \text{ uf}) \text{ } x \text{ } y$

where  $\text{per-union } R \text{ } x \text{ } y$  is the equivalence relation that results from merging the respective equivalence classes in the relation  $R$  that  $x$  and  $y$  belong to.

But what happens if  $x$  or  $y$  is not an element of the partial equivalence relation  $R$ ? In that case, the equivalence relation is unchanged, which means that  $\text{per-union } R \text{ } x \text{ } y = R$ . This, however, can be seen as a misuse of the UF data structure, since we initialise it with a fixed set of elements  $A$  and expect the user to only work with these elements. Therefore, we introduce the following definitions that characterise valid union(s) with regard to this initial set.

```
valid-union :: ufa ⇒ nat ⇒ nat ⇒ bool
valid-union uf a b = a ∈ Field (ufa-α uf) ∧ b ∈ Field (ufa-α uf)

valid-unions :: ufa ⇒ (nat × nat) list ⇒ bool
valid-unions uf us = ∀(x, y)∈set us. valid-union uf x y
```

### 3 Simple Certifying Union-Find Algorithm

Building on the UF ADT, we now develop a simple EXPLAIN operation that, for a given list of equations  $us::('a \times 'a) \text{ list}$ , takes two elements  $x$  and  $y$  and produces a certificate that  $x = y$  modulo  $us$ . The certificate is given in terms of a data type  $\text{eq-prf}$  with its corresponding system  $\vdash_$  of inference rules as seen in Fig. 1. As expected, we have inference rules that utilise the reflexivity, symmetry, and transitivity of equality as well as an assumption rule. To improve readability, we use the infix operator  $\triangleright$  to denote the proof term for transitivity.

We prove that  $\vdash_$  is sound and complete with respect to the equivalence relation induced by  $us$ , i.e. the equivalence closure of  $us$ . In Isabelle, we define

$$\begin{array}{ll} \text{symcl} :: 'a\ rel \Rightarrow 'a\ rel & \text{equivcl} :: 'a\ rel \Rightarrow 'a\ rel \\ \text{symcl } r = r \cup r^{-1} & \text{equivcl } r = (\text{symcl } r)^* \end{array}$$

and prove the theorem below.

**Theorem 1 (Soundness and Completeness of  $\vdash_=_$ ).** *If  $us \vdash_=_ p : (x, y)$  then  $(x, y) \in \text{equivcl}(\text{set } us)$ . Conversely, If  $(x, y) \in \text{equivcl}(\text{set } us)$  then  $\exists p. us \vdash_=_ p : (x, y)$ .*

Our goal is to implement the EXPLAIN operation using a UF data structure, so we fix an initial UF data structure  $uf$ . For a list of equations  $us$  or, in terms of the UF data structure, UNION operations, the current state of the UF data structure is then equal to ufa-unions  $uf\ us$  where we define

$$\begin{aligned} \text{ufa-unions} :: ufa \Rightarrow (\text{nat} \times \text{nat})\ list \Rightarrow ufa \\ \text{ufa-unions} = \text{foldl } (\lambda uf\ (x, y). \text{ufa-union } uf\ x\ y). \end{aligned}$$

Here, we require the unions  $us$  to be valid unions with respect to  $uf$ . Moreover, it must hold that  $\text{ufa-}\alpha\ uf \subseteq \text{Id}$  because the only way to justify an equality from an empty list of equations using  $\vdash_=_$  is by reflexivity. Finally, we also constrain  $us$  to be *effective* unions meaning that no union shall be redundant with respect to the unions preceding it. Note that redundant unions have no effect on the state of the UF data structure anyways so there is no need to record them. We formalise effectiveness with the following definitions.

$$\begin{aligned} \text{eff-union} :: ufa \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \\ \text{eff-union } uf\ a\ b = \text{valid-union } uf\ a\ b \wedge \text{ufa-rep-of } uf\ a \neq \text{ufa-rep-of } uf\ b \\ \text{eff-unions} :: ufa \Rightarrow (\text{nat} \times \text{nat})\ list \Rightarrow \text{bool} \\ \text{eff-unions } uf\ [] = \text{True} \\ \text{eff-unions } uf\ ((a, b) \# us) = \\ \quad \text{eff-union } uf\ a\ b \wedge \text{eff-unions } (\text{ufa-union } uf\ a\ b)\ us \end{aligned}$$

Similarly to  $ufa$ , we encapsulate pairs  $(uf, us)$  that are well-formed with respect to the constraints above by an ADT  $ufe$ . We choose this simple representation of the UFE data structure to ease formal reasoning, while a more efficient implementation is described in Section 5.2.

$$\text{typedef } ufe = \{(uf, us) \mid \text{ufa-}\alpha\ uf \subseteq \text{Id} \wedge \text{eff-unions } uf\ us\}$$

$$\frac{i < |us| \quad us ! i = (x, y)}{us \vdash_=_ \text{AssmP } i : (x, y)} \quad \frac{}{us \vdash_=_ \text{ReflP } x : (x, x)} \\ \frac{us \vdash_=_ p : (x, y)}{us \vdash_=_ \text{SymP } p : (y, x)} \quad \frac{us \vdash_=_ p_1 : (x, y) \quad us \vdash_=_ p_2 : (y, z)}{us \vdash_=_ p_1 \vee p_2 : (x, z)}$$

Figure 1: The system of inference rules  $\vdash_=_$  on the data type *eq-prf* of proofs. We write  $us \vdash_=_ p : (x, y)$  to say that  $p$  proves  $x = y$  assuming the equalities  $us$ .

We lift operations on such pairs  $(uf, us)$  to obtain (1)  $\text{unions} :: ufe \Rightarrow (\text{nat} \times \text{nat}) \text{ list}$ , (2)  $\text{uf-ds} :: ufe \Rightarrow ufa$ , (3)  $\text{ufe-init} :: \text{nat} \Rightarrow ufe$ , and (4) both  $\text{ufe-union} :: ufe \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow ufe$  and its dual (5)  $\text{rollback} :: ufe \Rightarrow ufe$ . The meaning of these operations is the following: (1)  $\text{unions } ufe$  is the list of unions  $us$ , (2)  $\text{uf-ds } ufe$  represents the current state of the UF data structure, i.e.  $\text{ufa-unions } uf \ us$ , (3)  $\text{ufe-init } n$  initialises the data structure with  $n$  elements and an empty list of unions, (4)  $\text{ufe-union } ufe \ a \ b$  appends an effective union  $(a, b)$  to  $us$ , and (5)  $\text{rollback } ufe$  removes the last union from  $us$ . Furthermore, we lift the remaining operations on  $ufa$  to  $ufe$  via  $\text{uf-ds}$ , replacing the prefix  $\text{ufa}$  by  $\text{ufe}$ . For example, we lift  $\text{ufa-rep-of}$  by letting  $\text{ufe-rep-of } ufe \equiv \text{ufa-rep-of } (\text{uf-ds } ufe)$ .

```

explain :: ufe ⇒ nat ⇒ nat ⇒ nat eq-prf
explain ufe x y =
  if unions ufe = [] then RefIP x
  else let ufe0 = rollback ufe; (a, b) = last (unions ufe);
       a-b-P = AssmP |unions ufe0|
    in if ufe-rep-of ufe0 x = ufe-rep-of ufe0 y then explain ufe0 x y
      else if ufe-rep-of ufe0 x = ufe-rep-of ufe0 a
        then explain ufe0 x a ∇ a-b-P ∇ explain ufe0 b y
        else explain ufe0 x b ∇ SymP a-b-P ∇ explain ufe0 a y

explain-partial :: ufe ⇒ nat ⇒ nat ⇒ nat eq-prf option
explain-partial ufe x y =
  if (x, y) ∈ equivcl (set (unions ufe)) then Some (explain ufe x y)
  else None

```

Figure 2: A simple implementation of the EXPLAIN operation.

At last, we implement the EXPLAIN operation as depicted in Fig. 2. The algorithm assumes that the given elements  $x$  and  $y$  are equal modulo unions  $ufe$ .

If  $\text{unions } ufe = []$ , then  $x$  and  $y$  must be equal which we certify with  $\text{RefIP } x$ .

Otherwise, we distinguish between two cases: (1) The elements  $x$  and  $y$  are already equal modulo unions ( $\text{rollback } ufe$ ), so we proceed recursively with  $\text{rollback } ufe$ . (2) In the case where the most recent equation  $a = b$  is necessary for  $x = y$  to hold, we either have  $x = a$  and  $b = y$  or  $x = b$  and  $a = y$  modulo unions ( $\text{rollback } ufe$ ). Assuming the former holds — the other case is symmetric — we recursively construct the certificates for  $x = a$  and  $b = y$ . Together with the assumption  $a = b$ , we obtain  $x = y$  by transitivity. The termination of  $\text{explain}$  is easily proven because the length of  $\text{unions } ufe$  decreases in each recursive call. Dually, this termination argument gives rise to the following induction principle.

**Lemma 1 (Induction on  $ufe$ ).** *In order to prove  $P \ ufe$  for all  $ufe$ , we have two inductive cases, both fixing an arbitrary  $ufe$ : (1) Assume  $\text{ufe-}\alpha \ ufe \subseteq \text{Id}$  as well as  $\text{unions } ufe = []$  and show  $P \ ufe$ . (2) Assume  $\text{eff-union } (\text{uf-ds } ufe) \ a \ b$  as well as  $P \ ufe$  and show  $P \ (\text{ufe-union } ufe \ a \ b)$ .*

We condense the intuition above into the completeness theorem below, which we prove using the induction principle from Lemma 1.

**Theorem 2 (Completeness of explain).** *If  $(x, y) \in \text{equivcl}(\text{set}(\text{unions } ufe))$  then  $\text{unions } ufe \vdash_=_ \text{explain } ufe x y : (x, y)$ .*

The `explain` function is not sound, though. This is because it always returns a certificate, even if  $x$  and  $y$  are not equal modulo  $us$ . To account for this case, we wrap `explain` into a partial function `explain-partial` (cf. Fig. 2) that fails if  $x = y$  is not provable. Soundness and completeness can then be lifted from the soundness of  $\vdash_=$  and the completeness of `explain`, respectively. Note that membership of `equivcl` can actually be implemented using UF operations as the following lemma demonstrates. Moreover, it holds that  $x \in \text{Field}(\text{ufa-}\alpha ufe) \iff x < n$  where  $n$  is the length of the list representing  $ufe$ .

**Lemma 2.** *We have  $(x, y) \in \text{equivcl}(\text{set}(\text{unions } ufe))$  iff  $x = y \vee x \in \text{Field}(\text{ufe-}\alpha ufe) \wedge y \in \text{Field}(\text{ufe-}\alpha ufe) \wedge \text{ufe-rep-of } ufe x = \text{ufe-rep-of } ufe y$ .*

## 4 Efficient Certifying Union-Find Algorithm

In the previous section, we developed an EXPLAIN operation that iteratively removes the most recent union from a list of unions, identifying which of them, when viewed as equalities, are necessary to prove the input arguments equal. Iterating through all equalities seems inefficient, though. Intuitively, we aim to return only those on the path between the arguments, viewing the equalities as an undirected graph. To realise this, Nieuwenhuis and Oliveras [20] use a UF data structure represented as forest of rooted trees as described in Section 2.1. They modify the data structure such that, for each union between  $a$  and  $b$ , the newly added edge in the forest gets annotated with  $(a, b)$ . To understand why this allows for a more efficient implementation of the EXPLAIN operation, suppose that we want to certify that  $x$  is equal to  $y$ . Clearly, only the edges of the subtree rooted at the lowest common ancestor (LCA) of  $x$  and  $y$ , as illustrated in Fig. 3, are relevant to explain why  $x$  is equal to  $y$ . Furthermore, let  $(a, b)$  be the most recent union on either of the paths from the LCA to  $x$  or  $y$ . Here, we assume w.l.o.g. that  $(a, b)$  is on the path to  $x$ . The corresponding edge separates the tree rooted at the LCA into two subtrees as indicated by the patterns, one containing  $a$  and the other one  $b$ . Moreover, the paths from the LCA can't overlap, so  $x$  and  $y$  also belong to different subtrees. Ultimately, to certify the equality of  $x$  and  $y$ , we recursively prove that  $x$  is equal to  $a$  and  $y$  to  $b$ . Then, we put everything together using transitivity and the equality  $a = b$ . This terminates since  $(a, b)$  is the most recent union and we only consider less recent unions in the recursive steps. All in all, this gives a  $\mathcal{O}(k \log n)$  EXPLAIN operation on a UF data structure with union-by-size, where  $k$  is the number of unions required for an explanation [20]. This is an improvement over the naive algorithm where we iterate over all (up to  $n - 1$ ) unions.

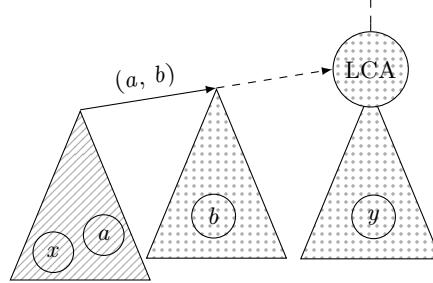


Figure 3: For arguments  $x$  and  $y$ ,  $\text{explain}'$  considers an edge annotated with  $(a, b)$  that separates the subtree rooted at the LCA of  $x$  and  $y$  into two subtrees: one containing  $x$  and  $a$  and the other one containing  $y$  and  $b$ .

To achieve optimal almost constant running time for UNION and FIND, we need path compression in addition to union-by-size. Path compression, however, is incompatible with the EXPLAIN operation, so Nieuwenhuis and Oliveras [20] propose to maintain two copies of the UF data structure, one with and one without path compression.

#### 4.1 Implementation

To obtain an efficient EXPLAIN operation, we leverage the structure of the UF data structure, which is a forest of rooted trees. We make this structure accessible by defining a function  $\text{ufa-parent-of} :: \text{ufa} \Rightarrow \text{nat} \Rightarrow \text{nat}$  via lifting, where  $\text{ufa-parent-of } uf\ x$  returns the parent of  $x$ . This function is related to  $\text{ufa-rep-of}$  in the obvious way, i.e. we have  $\text{ufa-parent-of } uf\ x = x$  iff  $\text{ufa-rep-of } uf\ x = x$  for  $x \in \text{Field}(\text{ufa-}\alpha\ uf)$ . With this, we formalise the concept of UFE forests, define the notion of associated unions within this forest, and introduce the two auxiliary functions that are the ingredients to the efficient EXPLAIN operation.

**UFE forests** It is often useful to view the forest of rooted trees underpinning the UF data structure as a graph. For this purpose, we use the graph theory library [25] due to Noschinski, which is available as an entry of the AFP [24]. The library allows us to represent a graph as a record with the fields `verts` and `arcs` for its vertices and edges, where edges are pairs of vertices. The forest induced by a UF data structure is then defined as follows.

```

 $\text{ufa-forest-of } uf =$ 
  let  $vs = \text{Field}(\text{ufa-}\alpha\ uf)$ 
  in ( $\text{verts} = vs$ ,
        $\text{arcs} = \{\text{ufa-parent-of } uf\ x, x) \mid x \in vs \wedge \text{ufa-parent-of } uf\ x \neq x\}$ )
 $\text{ufe-forest-of } ufe \equiv \text{ufa-forest-of } (\text{uf-ds } ufe)$ 

```

Note that we choose (somewhat arbitrarily) to direct the edges away from the root because it aligns more naturally with the notion of a directed rooted tree. Additionally, this choice ensures compatibility with the *directed-forest* locale, which we implemented on top of the graph library. For brevity, we omit the details here and direct the reader to the formalisation, but suffice it to say that typical properties of forests, e.g. the absence of cycles, are proved in this locale. To collect facts that are specific to UF forests, we define a locale *ufa-forest* fixing a UF data structure *uf*. In the context of this locale, we show that *ufa-forest-of uf* fulfils the requirements of a *directed-forest*, meaning that the facts in the latter locale transfer over to the former. Similarly, we introduce the locale *ufe-forest* fixing a UFE data structure *ufe*, where *uf-ds ufe* is a *ufa-forest*.

**Associated unions** As illustrated by Fig. 3, we annotate each edge of the UFE forest with the union that caused its creation, i.e. for an effective union  $(a, b)$ , we annotate the newly created edge  $e$  between the *ufe-rep-of ufe a* and *ufe-rep-of ufe b* with  $(a, b)$ . We say that  $(a, b)$  is the *associated union* of  $e$ . Since the underlying UF data structure is expressed in terms of parent pointers, we actually associate the union  $(a, b)$  with *ufe-rep-of ufe a*. Furthermore, we use an index into unions *ufe* rather than storing the union  $(a, b)$  directly. This concept is formalised in the constant *au-ds :: ufe ⇒ nat ⇒ nat option* whose specific implementation we skip over here; instead, we only state its characteristic properties:

- If unions *ufe* = [] then *au-ds ufe* = ( $\lambda x$ . None).
- For an effective union  $(a, b)$ , i.e if we have eff-union (*uf-ds ufe*)  $a\ b$ , it holds that *au-ds (ufe-union ufe a b)* = (*au-ds ufe*)(*ufe-rep-of ufe a*  $\mapsto$  |unions *ufe*|), where  $(f(x \mapsto y))\ z$  = (if  $z = x$  then Some  $y$  else  $f z$ ).

**Determining the LCA in a UFE forest** The first auxiliary functions lists the elements on the path from the representative to some element. Similarly to *ufa-rep-of*, this function is only well-defined for elements  $x \in \text{Field } (\text{ufa-}\alpha\ \text{uf})$  of a given UF data structure *uf*. Now, let *px* be the path from the representative of  $x$  to  $x$  and *py* be the path from  $y$ 's representative to  $y$ . Then, every element of a common prefix of *px* and *py* is a common ancestor of  $x$  and  $y$  and the LCA is exactly the last element of the longest common prefix of *px* and *py*.

```

awalk-verts-from-rep :: ufa ⇒ nat ⇒ nat list
awalk-verts-from-rep uf x =
  let px = ufa-parent-of uf x
  in if px = x then [x] else awalk-verts-from-rep uf px @ [x]

ufa-lca :: ufa ⇒ nat ⇒ nat ⇒ nat
ufa-lca uf x y =
  let px = awalk-verts-from-rep uf x; py = awalk-verts-from-rep uf y
  in last (longest-common-prefix px py)

```

Again, we abbreviate *ufe-lca ufe*  $\equiv$  *ufa-lca (uf-ds ufe)*. It holds that *ufa-lca* is indeed an LCA provided that the arguments share the same representative and

thus are in the same tree of the forest. For brevity, we omit the definition of `lca` here and refer to the formalisation instead.

**Lemma 3.** *If  $\{x, y\} \subseteq \text{Field } (\text{ufa-}\alpha \text{ uf})$  and  $\text{ufa-rep-of uf } x = \text{ufa-rep-of uf } y$ , then `lca` (`ufa-forest-of uf`) (`ufa-lca uf x y`)  $x y$ .*

We later prove key properties of EXPLAIN using the induction principle from Lemma 1, making it essential to understand how the auxiliary functions behave under effective unions. The lemma below shows that `ufa-lca` is invariant under a union  $(a, b)$  if its arguments share the same representative beforehand. Otherwise, the union introduces an edge from the representative of  $a$  to that of  $b$ , connecting the trees that  $x$  and  $y$  belong to at their respective roots. Due to the orientation of this new edge, we know that the LCA of  $x$  and  $y$  must be the representative of  $b$  after performing the union.

**Lemma 4.** *Assume `eff-union uf a b` and  $\{x, y\} \subseteq \text{Field } (\text{ufa-}\alpha \text{ uf})$ . If  $\text{ufa-rep-of } (\text{ufa-union uf } a b) x = \text{ufa-rep-of } (\text{ufa-union uf } a b) y$  then `ufa-lca` (`ufa-union uf a b`)  $x y = (\text{if ufa-rep-of uf } x = \text{ufa-rep-of uf } y \text{ then ufa-lca uf } x y \text{ else ufa-rep-of uf } b)$ .*

**Finding the most recent union on a path** For the second auxiliary function, we walk the path from the second argument  $x$  to the first argument  $y$  and return the most recent associated union, i.e. the maximum index with respect to unions `ufe` on that path. In Isabelle, we define the following function.

```
find-newest-on-path :: ufe ⇒ nat ⇒ nat ⇒ nat option
find-newest-on-path ufe y x =
  if y = x then None
  else max (au-ds ufe x) (find-newest-on-path ufe y (ufe-parent-of ufe x))
```

As explained earlier, we only use this function on an element in conjunction with its LCA relative to another element. Thus, there is a path between the two arguments and the function is well-defined for such inputs. The path, however, can be empty, in which we return `None`, making the function partial.

As before, we are interested in how the function behaves under effective unions. Since unions only join trees at their roots, existing paths in the tree are unchanged by unions, so, for elements in the same equivalence class, the function is invariant under unions. If, on the other hand, two elements only become part of the same equivalence class as a result of a union  $(a, b)$ , then  $(a, b)$  must be on the path between those elements and, as it is the most recent union, the function returns the index of that union.

**Lemma 5.** *Assume that `eff-union (uf-ds ufe) a b` and  $y$  is reachable from  $x$  in `ufe-forest-of` (`ufe-union ufe a b`), then it holds that `find-newest-on-path` (`ufe-union ufe a b`)  $x y = (\text{if ufe-rep-of ufe } x = \text{ufe-rep-of ufe } y \text{ then find-newest-on-path ufe } x y \text{ else Some } |\text{unions ufe}|)$ .*

```

explain' :: ufe ⇒ nat ⇒ nat ⇒ nat eq-prf
explain' ufe x y =
  if x = y then ReflP x
  else let lca = ufe-lca ufe x y;
    newest-x = find-newest-on-path ufe lca x;
    newest-y = find-newest-on-path ufe lca y
    in if newest-y ≤ newest-x
      then let (ax, bx) = unions ufe ! the newest-x
        in explain' ufe x ax ▽ AssmP (the newest-x) ▽
          explain' ufe bx y
      else let (ay, by) = unions ufe ! the newest-y
        in explain' ufe x by ▽ SymP (AssmP (the newest-y)) ▽
          explain' ufe ay y

```

Figure 4: Efficient version of the EXPLAIN operation.

**Explain** With the auxiliary functions in place, we are set to implement the efficient EXPLAIN operation as shown in Fig. 4.

Given arguments  $x$  and  $y$ , we first check whether they are equal and, if so, we justify their equality by reflexivity.

Otherwise, we determine the LCA of the two elements and the most recent associated union on both of the paths from the elements to the LCA. Note that, if the LCA is equal to  $x$  or to  $y$ , the respective path to the LCA is empty; nevertheless, it is impossible that both  $x$  and  $y$  are equal to the LCA because we are in the case where  $x \neq y$ . Consider, for the sake of an explanation, the case where the most recent union  $(ax, bx)$  is on the path to  $x$ . This means, as illustrated in Fig. 3, that  $x$  and  $ax$  as well as  $y$  and  $bx$  are in the same subtree, respectively. Thus, we call `explain'` recursively and, using transitivity, combine the resulting proofs of  $x = ax$  and  $bx = y$  with the assumption that  $ax = bx$ .

The last case, where the most recent union is on the path from  $y$  to the LCA, is symmetric, which, accordingly, requires us to apply the symmetry rule after using the assumption rule on the most recent union.

As we will show below, `explain'` only terminates for specific inputs. The domain on which the function is well-defined is again characterised by a domain predicate `explain'-dom :: ufe ⇒ nat × nat ⇒ bool`.

## 4.2 Correctness

Verifying the functional correctness of `explain'` requires proving termination as well as soundness and completeness. We prove termination directly, while we obtain soundness and completeness by showing extensional equality of `explain'` and `explain`. As `explain'`, like `explain`, does not validate its input, we assume for the remainder of this section that (1)  $\{x, y\} \subseteq \text{Field}(\text{ufe-}\alpha \text{ ufe})$  and (2) `ufe-rep-of ufe x = ufe-rep-of ufe y`.

To establish termination of  $\text{explain}'$ , we first prove that termination remains invariant under an effective union using the invariance of  $\text{find-newest-on-path}$  and  $\text{ufe-lca}$  under an effective union (see Lemmas 4 and 5). From this, the termination of  $\text{explain}'$  follows by induction on  $\text{ufe}$ .

**Lemma 6.** *Assume  $\text{explain}'\text{-dom } \text{ufe } (x, y)$  and  $\text{eff-union } (\text{uf-ds } \text{ufe}) \ a \ b$ , then it holds that  $\text{explain}'\text{-dom } (\text{ufe-union } \text{ufe } a \ b) \ (x, y)$ .*

**Theorem 3 (Termination).**  $\text{explain}'\text{-dom } \text{ufe } (x, y)$

By Theorem 3 and the invariance of the auxiliary functions under effective unions, we deduce that  $\text{explain}'$  is also invariant under effective unions.

**Lemma 7.** *If  $\text{eff-union } (\text{uf-ds } \text{ufe}) \ a \ b$  then  $\text{explain}' \ (\text{ufe-union } \text{ufe } a \ b) \ x \ y = \text{explain}' \ \text{ufe } x \ y$ .*

Given the definition of  $\text{explain}$ , we now understand the behaviour of both  $\text{explain}$  and  $\text{explain}'$  under effective unions. Thus we conclude, by induction on  $\text{ufe}$ , that  $\text{explain}$  is extensionally equal to  $\text{explain}'$ .

**Theorem 4 (Correctness).**  $\text{explain } \text{ufe } x \ y = \text{explain}' \ \text{ufe } x \ y$

## 5 Refinement to an Efficiently Executable Specification

In the previous section, we described a refined recursion scheme for EXPLAIN that avoids iterating through all input equalities. To turn this into an efficiently executable specification, we refine two aspects of the UFE data structure.

First, we employ the union-by-size heuristic [9], i.e. we always attach the tree with fewer elements to the one with more elements during a UNION. This ensures that all trees in the UF data structure have height at most  $\mathcal{O}(\log n)$  where  $n$  is the number of elements of the data structure. This yields  $\mathcal{O}(\log n)$  running time for UNION and FIND as well as  $\mathcal{O}(k \log n)$  for EXPLAIN.

Then, we take this functional UFE data structure and refine it to an imperative specification, thereby giving a concrete implementation. In doing that, we are careful to refine lists by arrays, guaranteeing constant time access to e.g. the parent of an element in the UF data structure. Additionally, we maintain a copy of the UF data structure with path compression as described in Section 4, improving the performance of UNION and FIND to almost constant running time.

### 5.1 Union-by-size Heuristic

As mentioned in Section 2.2, our formalisation of the UF data structure extends a formalisation by Lammich [15, 16]. The latter formalisation already introduces the union-by-size heuristic, but it does so during the refinement to Imperative HOL. We raise the union-by-size heuristic to the purely functional level of HOL, which lets us exploit Isabelle's lifting and transfer infrastructure [12]. In addition,

we introduce another optimisation: we represent the UF data structure as a single list of integers, eliminating the data structure recording the size information.

As a prerequisite for the union-by-size heuristic, we define a function that determines the equivalence class of an element  $x$  in the data structure  $uf$ . More specifically, we use the relational image operator (“ $\circ$ ”) on the equivalence relation  $ufa\text{-}\alpha uf$  to obtain all the elements that are equivalent to  $x$ . The associated size of an element is then the cardinality of its equivalence class.

$$\begin{array}{ll} \text{ufa-eq-class} :: ufa \Rightarrow nat \Rightarrow nat \text{ set} & \text{ufa-size} :: ufa \Rightarrow nat \Rightarrow nat \\ \text{ufa-eq-class } uf\ x = \text{ufa-}\alpha\ uf\ “\{x\} & \text{ufa-size } uf\ x = |\text{ufa-eq-class } uf\ x| \end{array}$$

With this, we perform the UNION operation such that the element with the smaller size is always passed as the first argument. The underlying implementation of the data structure always updates the parent pointer of the representative of the first argument to the representative of the second argument, thus yielding a UNION operation that attaches smaller trees in the UF forest to larger trees.

$$\begin{aligned} \text{ufa-union-size} :: ufa \Rightarrow nat \Rightarrow nat \Rightarrow ufa \\ \text{ufa-union-size } ufa\ x\ y = \\ \quad \text{let } rep\text{-}x = \text{ufa-}\text{rep-of } ufa\ x; rep\text{-}y = \text{ufa-}\text{rep-of } ufa\ y \\ \quad \text{in if } \text{ufa-size } ufa\ rep\text{-}x < \text{ufa-size } ufa\ rep\text{-}y \text{ then ufa-union } ufa\ x\ y \\ \quad \quad \text{else ufa-union } ufa\ y\ x \end{aligned}$$

Looking closely at the definition, we see that  $\text{ufa-size}$  is only ever used on the representative of an element. Moreover, in the representation of  $ufa$  as a list of natural numbers, the representatives are exactly those where the parent pointer is self-referential. Ultimately, we integrate both insights and encode the UF data structure as an ADT  $ufsi$ , which is implemented by a list of integers: we use a negative number to indicate that a parent pointer is self-referential, using the absolute value of the number as the size at the same time. The other parent pointers are encoded as non-negative numbers as before.

## 5.2 From Functional to Imperative Specification

To obtain an imperative specification, we formulate a refined version of the EXPLAIN operation in the heap monad provided by the Imperative HOL [5] framework. This framework comes with an extension to Isabelle’s code generator allowing us to generate imperative code in several target languages including Standard ML. Since Imperative HOL only comes with limited capabilities to analyse programs in its heap monad, we bring in Lammich’s [15] separation logic framework for Imperative HOL. The framework lets us reason about the state of the heap using heap assertions, which describe data stored on the heap and their properties. All our data structures are ultimately represented as arrays on the heap, so we ensure with heap assertions that the content of the arrays represents our data structures throughout the operations we perform on them.

With the automation provided by Lammich’s framework, it is straightforward to implement the operations and prove their correctness. The process is similar

to the refinement of the UF data structure [15]. Thus, we forgo a discussion of how individual functions are refined and refer to the formalisation instead.

The only remaining noteworthy detail is the representation of the UFE data structure in Imperative HOL. Our implementation consists of a UF data structure, a partial function recording the associated union of each parent pointer, and the chronological list of unions. The UF data structure is represented as an array of integers. For the associated unions, we use an array of options to represent the partial function. This works as the domain is actually fixed, i.e. the domain of the partial function is exactly the elements of the UF data structure, which, in our case, are the natural numbers up to some fixed  $n$ . Lastly, we represent the list of unions as a dynamic array using the type *array-list*. The type wraps an array together with a natural number indicating how many cells of the array, counting from the first position, are occupied. We can then grow the array dynamically by pushing elements to the end, doubling its size each time it becomes fully occupied. Hence, we achieve amortised constant running time for adding new unions and constant time random access, which are the operations required by the EXPLAIN operation. We assemble these components into a record type *ufe-imp*. Finally, we extend *ufe-imp* with a UF data structure with path compression, thus obtaining the record type *ufe-c-imp*.

We define a heap assertion  $\text{is-ufe} :: \text{ufe} \times \text{nat} \Rightarrow \text{ufe-imp} \Rightarrow \text{assn}$  that relates instances of the ADT *ufe* with instances of *ufe-imp*. The assertion just relates the components of *ufe-imp* with the corresponding functions on *ufe*, so we omit it for brevity. The only aspect requiring further explanation is the natural number  $n$  in the first argument. Its purpose is to ensure that the elements of the initial UF data structure and the domain of the associated unions are both the numbers up to  $n$ . To obtain the assertion  $\text{is-ufe-c} :: \text{ufe} \times \text{nat} \Rightarrow \text{ufe-c-imp} \Rightarrow \text{assn}$ , we additionally require that the representatives in the UF data structure with path compression corresponds to the representatives in the UFE data structure.

Again, refining the operations on *ufe-c-imp* is routine; so, we only show the final correctness theorem for *explain-partial-imp*, an imperative version of *explain'* that ensures soundness following the approach of *explain-partial* in Section 3.

**Theorem 5.** *We prove the following Hoare triple, which entails total correctness in the Separation Logic Framework [16]:*  $\langle \text{is-ufe-c } (\text{ufe}, n) \text{ ufe-c-imp} \rangle \text{ explain-partial-imp } \text{ufe-c-imp } x \text{ } y \langle \lambda r. \text{ is-ufe-c } (\text{ufe}, n) \text{ ufe-c-imp } * \uparrow (r = \text{explain-partial } \text{ufe } x \text{ } y) \rangle_t$

## 6 Benchmarking the Exported Code

In the previous section, we obtained an executable imperative specification of the UFE data structure, from which we can export code to functional target languages while utilising their respective support for imperative programming like destructive array updates in Standard ML (SML). This raises the question whether exporting imperative code to a functional language is a good fit, performance wise. In addition, SMT solvers are usually implemented in imperative

language such as C++. Therefore, we compare the exported SML code against a hand-written C++ implementation of the executable specification.

We analyse the performance on two test cases: (1) in the former case, the number of proof steps for an EXPLAIN operation is linear in the number of elements but the depth of the UF forest is constant, (2) while in the latter, the depth of the UF forest as well as the number of proof steps is logarithmic in the number of elements. For both cases, we choose a natural number  $n$ , initialise the UFE data structure with  $2^n$  elements, perform UNION operations that results in the desired UF forest, and finally perform a number (i.e. 1000 and 100000, respectively) of EXPLAIN operations with the arguments drawn from the uniform distribution over  $0, \dots, 2^n - 1$ . We identify the test cases by functions `wide` and `balanced`, which both have type  $\text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ list}$ . Fig. 5 illustrates the resulting UF forests or, more specifically, trees.

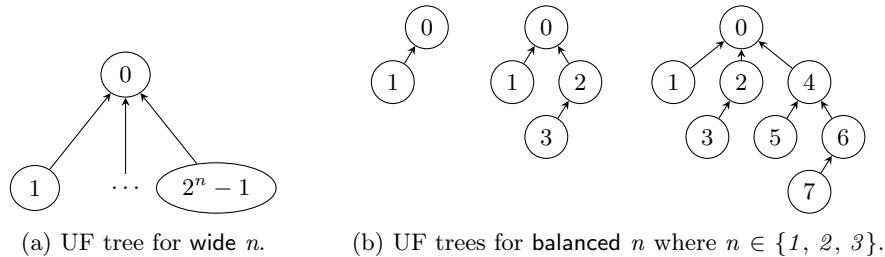


Figure 5: The UF trees resulting from performing the UNION operations with the arguments given by `wide` and `balanced`.

To perform our measurements, we compiled the exported Standard ML code with MLton<sup>8</sup> version 20210117, and the C++ code with g++<sup>9</sup> version 13.3.0. The results are shown in Table 1. The code export of Isabelle uses arbitrary sized integers to ensure soundness with respect to the executable specification while the C++ uses machine integers, so we also include a version of the exported code, annotated by `(Int)`, that uses machine integers.

The observed running time overhead of using arbitrary sized integers is roughly a factor of 2–2.5, matching that observed by Lammich and Tuerk [17]. The difference between SML with machine integers and C++ is roughly a factor of 2 for the UNION operations and a factor of 1.5 for the EXPLAIN operations throughout both test cases. The second test case exhibits some outliers: notably, between  $n = 23$  to  $n = 24$  for SML and between  $n = 24$  and  $n = 25$  for SML `(Int)`. This variance is due to garbage collection becoming a significant factor at large heap sizes, e.g. for  $n = 25$  the heap grows to above 5 gigabytes.

Overall, we found that employing a functional language results in a modest performance overhead when working with machine integers. We note that to

<sup>8</sup> <http://mlton.org>

<sup>9</sup> <https://gcc.gnu.org>

Impl.	18	19	20	21	22
SML	0.025/18.428	0.075/36.129	0.072/70.696	0.157/140.209	0.393/280.684
SML (Int)	0.011/8.341	0.011/16.249	0.024/29.695	0.051/62.442	0.092/131.532
C++	0.004/3.672	0.007/7.354	0.015/15.113	0.031/31.120	0.062/71.066

(a) Running times for `wide n`.

Impl.	22	23	24	25	26
SML	0.722/1.879	0.899/2.583	1.552/2.082	4.770/2.396	14.610/3.014
SML (Int)	0.174/0.750	0.227/0.781	0.695/0.900	2.474/0.920	2.785/1.027
C++	0.087/0.369	0.199/0.460	0.350/0.550	0.752/0.620	1.451/0.728

(b) Running times for `balanced n`.

Table 1: Wall-clock running times in seconds as measured on an Intel Core i7 4790k. For each  $n$ , we recorded the running time for performing the UNION operations and the EXPLAIN operations (separated by a slash).

soundly export such code, it would be necessary to change the element type of the UFE data structure from natural numbers to fixed-width words. This is plausible future work, as the number of elements is fixed for any instance of the data structure, and the only necessary operations on the elements are comparisons and indexing into arrays — operations that fixed-width words also support.

## 7 Conclusion and Future Work

We developed a formalisation of the UF data structure with an EXPLAIN operation based on a paper by Nieuwenhuis and Oliveras [20]. The formalisation includes a more naive version of the EXPLAIN operation than the one presented in the paper. We proved their equivalence as well as their soundness, completeness, and termination. Finally, we refined the functional representation of the data structure to an imperative one, allowing us to export efficient code.

In future work, we plan to verify the other variant of the UFE data structure as presented by Nieuwenhuis and Oliveras. This variant also forms the basis of their congruence closure algorithm, which is the logical next step. Ultimately, we want to work towards a verified, proof-producing version of the Nelson-Oppen algorithm [19] for the combination of theories.

**Acknowledgements** We thank Tobias Nipkow for reviewing a draft version of this paper and the anonymous referees for their thoughtful feedback.

**Disclosure of Interests** The authors have no competing interests to declare that are relevant to the content of this article.

## Bibliography

- [1] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974), ISBN 0-201-00029-6
- [2] Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Types for Proofs and Programs, pp. 34–50, Springer, Berlin, Heidelberg (2003), [https://doi.org/10.1007/978-3-540-24849-1\\_3](https://doi.org/10.1007/978-3-540-24849-1_3)
- [3] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 13243, pp. 415–442, Springer (2022), [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
- [4] Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: verit: An open, trustable and efficient SMT-solver. In: Conference on Automated Deduction, pp. 151–156, Springer, Berlin, Heidelberg (2009), [https://doi.org/10.1007/978-3-642-02959-2\\_12](https://doi.org/10.1007/978-3-642-02959-2_12)
- [5] Bulwahn, L., Krauss, A., Haftmann, F., Erköl, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics, pp. 134–149, Springer Berlin Heidelberg (2008), [https://doi.org/10.1007/978-3-540-71067-7\\_14](https://doi.org/10.1007/978-3-540-71067-7_14)
- [6] Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning **62**(3), 331–365 (2019), <https://doi.org/10.1007/s10817-017-9431-7>
- [7] Conchon, S., Filliâtre, J.C.: A persistent union-find data structure. In: Workshop on ML, p. 37–46, Association for Computing Machinery, New York, NY, USA (2007), <https://doi.org/10.1145/1292535.1292541>
- [8] Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchehkh, P.: Small proofs from congruence closure. In: Formal Methods in Computer-Aided Design, pp. 75–83, IEEE (2022), [https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2\\_13](https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_13)
- [9] Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Communications of the ACM **7**(5), 301–303 (1964), <https://doi.org/10.1145/364099.364331>
- [10] Guttmann, W.: Verifying the correctness of disjoint-set forests with Kleene relation algebras. In: Relational and Algebraic Methods in Computer Science, pp. 134–151, Springer International Publishing, Cham (2020), [https://doi.org/10.1007/978-3-030-43520-2\\_9](https://doi.org/10.1007/978-3-030-43520-2_9)
- [11] Hasbeck, M.P.L., Lammich, P.: Refinement with time - refining the runtime of algorithms in Isabelle/HOL. In: Interactive Theorem Proving, pp. 20:1–20:18 (2019), <https://doi.org/10.4230/LIPIcs.ITP.2019.20>

- [12] Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: Certified Programs and Proofs, pp. 131–146, Lecture Notes in Computer Science, Springer (2013), [https://doi.org/10.1007/978-3-319-03545-1\\_9](https://doi.org/10.1007/978-3-319-03545-1_9)
- [13] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Computer Aided Verification, pp. 1–35, Springer, Berlin, Heidelberg (2013), [https://doi.org/10.1007/978-3-642-39799-8\\_1](https://doi.org/10.1007/978-3-642-39799-8_1)
- [14] Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7**(1), 48–50 (1956), <https://doi.org/10.2307/2033241>
- [15] Lammich, P.: Refinement to imperative HOL. Journal of Automated Reasoning **62**(4), 481–503 (2017), <https://doi.org/10.1007/s10817-017-9437-1>
- [16] Lammich, P., Meis, R.: A separation logic framework for imperative HOL. Archive of Formal Proofs (2012), ISSN 2150-914x, [https://isa-afp.org/entries/Separation\\_Logic\\_Imperative\\_HOL.html](https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html), Formal proof development
- [17] Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft’s algorithm. In: Interactive Theorem Proving, pp. 166–182, Springer, Berlin, Heidelberg (2012), [https://doi.org/10.1007/978-3-642-32347-8\\_12](https://doi.org/10.1007/978-3-642-32347-8_12)
- [18] de Moura, L.M., Bjørner, N.S.: Proofs and refutations, and Z3. In: International Workshop on the Implementation of Logics, CEUR Workshop Proceedings, vol. 418, CEUR-WS.org (2008), URL <https://ceur-ws.org/Vol-418/paper10.pdf>
- [19] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979), <https://doi.org/10.1145/357073.357079>
- [20] Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Term Rewriting and Applications, pp. 453–468, Springer Berlin Heidelberg (2005), [https://doi.org/10.1007/978-3-540-32033-3\\_33](https://doi.org/10.1007/978-3-540-32033-3_33)
- [21] Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Information and Computation **205**(4), 557–580 (2007), <https://doi.org/10.1016/j.ic.2006.08.009>
- [22] Nipkow, T., Eberl, M., Haslbeck, M.P.L.: Verified textbook algorithms: A biased survey. In: Automated Technology for Verification and Analysis, p. 25–53, Springer, Berlin, Heidelberg (2020), [https://doi.org/10.1007/978-3-030-59152-6\\_2](https://doi.org/10.1007/978-3-030-59152-6_2)
- [23] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
- [24] Noschinski, L.: Graph theory. Archive of Formal Proofs (April 2013), ISSN 2150-914x, [https://isa-afp.org/entries/Graph\\_Theory.html](https://isa-afp.org/entries/Graph_Theory.html), Formal proof development
- [25] Noschinski, L.: A graph library for Isabelle. Mathematics in Computer Science **9**(1), 23–39 (2015), <https://doi.org/10.1007/S11786-014-0183-Z>

- [26] Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Conference on Automated Deduction, pp. 495–507, no. 11716 in LNAI, Springer (2019), [https://doi.org/10.1007/978-3-030-29436-6\\_29](https://doi.org/10.1007/978-3-030-29436-6_29)
- [27] Stevens, L., Nipkow, T.: A verified decision procedure for orders in Isabelle/HOL. In: Automated Technology for Verification and Analysis, pp. 127–143, Springer International Publishing, Cham (2021)
- [28] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. *J. ACM* **22**(2), 215–225 (1975), <https://doi.org/10.1145/321879.321884>
- [29] Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. *J. ACM* **31**(2), 245–281 (1984), <https://doi.org/10.1145/62.2160>